Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Discrete Algorithms 4 (2006) 106–141

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

Reducing structural changes in van Emde Boas' data structure to the lower bound for the dynamic predecessor problem ☆

George Lagogiannis [a], Christos Makris [a,b,*], Athanasios Tsakalidis [a,b]

[a] *Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece*
[b] *Computer Technology Institute, P.O. Box 1192, 26110 Patras, Greece*

Available online 28 January 2005

Abstract

We consider the problem of maintaining a dynamic ordered set of n integers in a universe U under the operations of insertion, deletion and predecessor queries. The computation model used is a unit-cost RAM, with a word length of w bits, and the universe size is $|U| = 2^w$. We present a data structure that uses $O(|U|/\log|U| + n)$ space, performs all the operations in $O(\log\log|U|)$ time and needs $O(\log\log|U|/\log\log\log|U|)$ structural changes per update operation. The data structure is a simplified version of the van Emde Boas' tree introducing, in its construction and functioning, new concepts, which help to keep the important information for searching along the path of the tree, in a more compact and organized way.

Keywords: Data structures; Worst case complexity; Predecessor problem; Lower bounds; Search trees

1. Introduction

The dynamic predecessor problem calls for maintaining a subset S of an ordered universe U under the following operations:

☆ We thank European Social Fund (ESF), Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS, for funding the above work.

* Corresponding author.
E-mail addresses: lagogian@ceid.upatras.gr (G. Lagogiannis), makri@ceid.upatras.gr (C. Makris), tsak@ceid.upatras.gr (A. Tsakalidis).

1570-8667/$ – see front matter
doi:10.1016/j.jda.2005.01.006

– *Insert*(e): inserts the element e into S,
– *Delete*(e): deletes the element e from S,
– *Pred*(e): returns the element $\max\{x \in S \mid x \leqslant e\}$.

Dynamic predecessor data structures were firstly described by van Emde Boas [14,15]. The proposed data structure uses $O(|U|)$ space and performs all the operations in $O(\log\log|U|)$ time. (A simplified description of this construction was given by Overmars [13].) Willard [16] gave a similar approach (V-trees) based on van Emde Boas' solution. V-trees use $O(|U|/\log|U| + n)$ space and perform the required operations in $O(\log\log|U|)$ time; here $n = |S|$ denotes the size of the stored set. The reduction in space complexity is due to the use of bucketing in the lower $\log\log|U|$ levels of the van Emde Boas' tree.

Fredman and Willard [9] described a linear space data structure that performs all operations in $O(\log n/\log\log n)$ time. Andersson [1,2] presented an $O(\sqrt{\log n})$ worst case time and linear space solution. Beame and Fich [5] managed to match the upper and lower bounds for the static predecessor problem. They presented a static data structure performing predecessor queries in $O(\min\{\log\log|U|/\log\log\log|U|, \sqrt{\log n/\log\log n}\})$ time and they also proved matching lower bounds. Combined with the exponential search trees [2,4], their data structure solves the dynamic predecessor problem in $O(\min\{\log\log n \log\log|U|/\log\log\log|U|, \sqrt{\log n/\log\log n}\})$ query and update time, using linear space. The attained time bound is probably not optimal, since it fails to match the $\Omega(\log\log|U|/\log\log\log|U|)$ lower bound of [5]. In this paper we present a data structure for the dynamic predecessor problem that uses $O(|U|/\log|U| + n)$ space, performs all operations in $O(\log\log|U|)$ worst case time and needs $O(\log\log|U|/\log\log\log|U|)$ structural changes per update operation. The data structure is another version of the van Emde Boas' tree and though it is more complex (both in theory and in practice) than the initial construction, we claim that its construction and functioning entail new concepts that could potentially help to devise a dynamic predecessor structure matching the lower bound of Beame and Fich's construction. We close by noting that the space complexity of our structure can be reduced to linear ($O(n)$), if we use dynamic perfect hashing [8]; in this case the predecessor query is still supported in $O(\log\log|U|)$ worst-case time, but the bounds for the update operations become amortized expected.

2. Preliminaries

The machine model used in this paper is the unit-cost RAM, with a word length of w bits. The universe U consists of the integers in the range $[0, 2^w - 1]$. It is assumed that the RAM can perform the standard AC^0 operations of addition, subtraction, comparison, bitwise operations and shifts, as well as multiplications in constant worst-case time on $O(w)$-bit operands. One of the basic features of the RAM is that the content of the elements are used for addressing, which is one of the basic differences with other comparison based models (such as *Pointer Machines*). The restriction to integers is not so important as it seems since besides the repeated use of integers in algorithmic problems, the ordering of objects belonging to most of the other basic data types is maintained if we perceive their

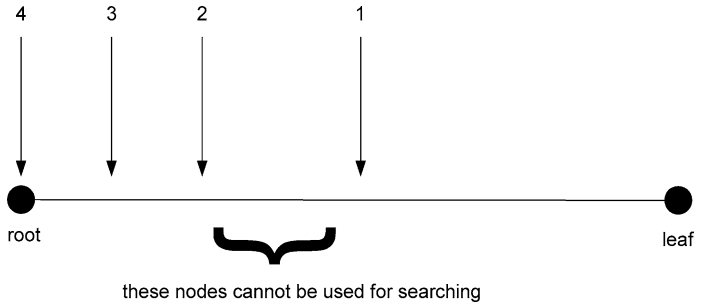

Fig. 1.

bit-string representations as integers; the most prominent example is the representation of floating point numbers in the IEEE 754 floating point standard (see also [3,4]).

In the following, we will describe a slight variation of the van Emde Boas' tree, as given in [13]. Assume w.l.o.g. that $|U| = 2^k$, for some k. The van Emde Boas' tree is a complete balanced binary tree T with depth $k = \log |U|$. It consists of an upper tree T_0 with depth $\lfloor k/2 \rfloor$, having approximately $\sqrt{|U|}$ leaves. Each of these leaves, corresponds to lower trees T_i, $i = 1, \ldots, \sqrt{|U|}$, which are of about $\sqrt{|U|}$ size. The same layout is recursively applied, in order to construct T_0 and each of the lower trees T_i. If we unfold the whole structure, then it consists of a total of $O(|U|)$ nodes. An ordered set S is stored in T by splitting it into subsets S_i, $i = 1, \ldots, \sqrt{|U|}$, each corresponding to the universe chunks for the lower trees T_i. Each S_i is stored in T_i as follows. If S_i contains less than three keys, it is stored in the root of T_i; otherwise the largest and the smallest elements are stored in the root of T_i and the rest is stored recursively in the lower trees. We store together with the root of T_i the size of S_i and if S_i is not empty we store i recursively in T_0. Moreover, an auxiliary doubly linked list is constructed, which stores each key's value, thus allowing the retrieval of its nearest neighbors in $O(1)$ time. It follows from the construction phase, that each node will not contain more than two keys. Since we have $O(|U|)$ nodes in the tree, this yields a total of $O(|U|)$ space requirement. The information along a path from the root to a leaf is stored in $\log \log |U|$ nodes, according to binary search steps. This means that for the retrieval/update of information, these $\log \log |U|$ nodes have to be accessed/updated. In fact, it can be proven that the structure can handle any update operation and the predecessor query in $O(\log \log |U|)$ time.

One should note that the van Emde Boas' tree, by storing information according to binary steps, denotes in advance that binary search for the retrieval has to be used. Since there is no known better way for searching along the path, this tactic is suitable. This observation is further explained in Fig. 1.

In Fig. 1 we depict the root to leaf path of van Emde Boas' solution, where the leaf stores the only element of the data structure. The first piece of information is stored in the middle of the root to leaf path (position 1); the second piece is stored in the middle of the root to the node 1 path (position 2) and so on. In order to reach position 4, we must first access positions 1, 2 and 3. This means that we have to use binary search.

Our main claim is that in order to achieve $o(\log \log |U|)$ query time we should not base our query procedure on the above update process; for example if an imaginary optimal

query procedure was to lead to an internal node between the nodes pointed at by the pointers 1 and 2 then we would not find there any information to continue our search. The above claim leads to the following requirement: *Suppose we want to locate predecessor*(x). *Let* u *be a node of the root to* x *path. If we access* u*, we must find there information, that could help us in our search directions*.

If this requirement holds, then the path is suitable for applying a better searching strategy, since the imaginary optimal query procedure has the ability, even implicitly, to potentially use *all* the nodes in the root to leaf path. In the next sections we will describe a data structure that meets this requirement. The information is stored in a continuous way and there are no $\log\log|U|$ nodes that have to be modified since we do not have to choose $\log\log|U|$ critical nodes along the path and store in them important information. The only information that characterizes a path, is the level that distinguishes two different kinds of nodes (*red* and *black*). In *every* node of the path there is information leading us to the direction of this critical level. After we have reached the level that distinguishes the red from the black nodes, our data structure requires only $O(\log\log|U|/\log\log\log|U|)$ time per update operation. Searching along the path still needs $O(\log\log|U|)$ time since a better strategy than binary search has not yet been found. But in our data structure the path is suitable for applying a better searching strategy, that would lead to an $o(\log\log|U|)$ time; in particular the formulation of such an optimal query strategy was the initial motivation for this work.

We conclude by stating that the proposed structure is mainly of theoretical interest since some of its main components are *q-heaps* [10,16] and so it is quite more complex (both theoretically and experimentally) than the initial van Emde Boas' tree.

3. The proposed data structure

Our structure is built on a complete binary trie with depth $\log|U| - \log\log|U|$. Each leaf defines a path from the root to that leaf. The elements are stored in auxiliary structures, called *leaf structures*, and each leaf structure is stored at special nodes of the trie. Let $v_1, v_2, \ldots, v_{\log|U|-\log\log|U|}$ be a path of the trie, where v_1 is the root. For that path, the following holds: $\exists i$, $1 \leqslant i \leqslant \log|U| - \log\log|U|$, such that for every node v_j that belongs to that path:

- If $j < i$ then the information stored in v_j is 1. In that case v_j is called *red* node.
- If $j > i$ then the information stored in v_j is 0. In that case v_j is called *black* node.
- If $j = i$ then the information stored in v_j is 1. v_j is a red node. Also, in v_j there is a pointer to a *leaf structure* and we say that v_j is an *end-node*.

Moreover, we demand that if a node v is red then either both children of v are red, or both children of v are black. The colors of the nodes can change dynamically according to the following operations:

- *Open*(v): (v is an end-node and u, w are the children of v. The nodes u and w are black). Node v is no longer an end-node, but remains a red node. u and w become end-nodes.

- *Close*(w, u): (w and u are red, their children are black and they are siblings). The nodes w and u become black and their common father becomes an end-node.

Let u be a node of the trie at depth i. Let $\langle p \rangle$ be the common prefix of all the leaves of the subtree rooted at u. If there is a pointer stored in u (u is an end-node), then this pointer leads to a leaf structure that contains elements with common prefix $\langle p \rangle$ and has a capacity of at most $\log\log|U| + i$ elements. All the end-nodes are connected through a doubly linked list, that is suitably maintained (in constant time) during the *Open*() and *Close*() operations. The depth of the trie is $\log|U| - \log\log|U|$ because we demand from a leaf structure to contain *at most* $\log|U|$ elements, which are the maximum number of elements that could be stored in the last $\log\log|U|$ levels of the tree. The leaf structure supports the following operations:

- $Pred_L(A, x)$: Locates the predecessor of x, within leaf structure A.
- $Insert_L(A, x)$: Inserts the element x, in the leaf structure A.
- $Delete_L(A, x)$: Deletes the element x, from the leaf structure A.
- $Union_L(A_1, A_2)$: Creates a new leaf structure A, that contains all the elements of A_1 and A_2.
- $Split_L(A, y)$: Splits leaf structure A into two new leaf structures, A_1 and A_2. A_1 contains all the elements of A that are smaller than y. A_2 contains the remaining elements of A.

We can now describe the operations in our data structure:

Pred(x)

1. Find the end-node of the root to x path (by performing binary search).
2. $Pred_L(A, x)$, where A is the leaf structure of that node. If x is the smallest element in A, then move to the previous end-node by using the doubly linked list. Let B be the leaf structure that this end-node points to; return the maximum element of B.

Insert(x)

1. Find the end-node of the path from the root to x (by performing binary search). Let w be that node and i be the depth of w.
2. $Insert_L(A, x)$, where A is the leaf structure of w.
3. If after the insertion the number of the stored elements becomes $\log\log|U| + i + 1$, then the following actions are performed:
   - *Open*(w).
   - $Split_L(A, y)$, where $y = \langle p \rangle 10 \ldots 0$ ($\langle p \rangle$ is the prefix of w). After the split two new leaf structures, A_1 and A_2, are created.
   - Store the pointer to A_1 into the left child of w.
   - Store the pointer to A_2 into the right child of w.
   - If A_1 or A_2 is empty perform some additional computation (the details are given later in the paper).

Delete(x)

1. Find the end-node of the path from the root to x (by performing binary search). Let w be that node and i be the depth of w. Let u be the brother of w.
2. $Delete_L(A, x)$ where A is the leaf structure of w.
3. If u is not an end-node exit; otherwise let Z be its respective leaf structure. If the sum of the elements of the leaf structures A and Z becomes $\log\log|U| + i - 1$ after the deletion of x, the following steps are performed:
   - $Close(u, w)$.
   - $Union_L(A, Z)$. A new leaf structure B is created.
   - Let r be the father of w. Store a pointer to B into r.

The worst-case time of the above operations consists of three components: (i) the time to access the last red node of a path (the end-node); (ii) the time to perform some "additional" computation (emerging from phase 3, last step of the *Insert* operation) that will be described later on; (iii) the time needed to handle a constant number of operations on the leaf structures.

The major contribution of our construction is to provide mechanisms that implement steps (ii) and (iii) in $O(\log\log|U|/\log\log\log|U|)$ time; while in the original data structure it would take $O(\log\log|U|)$ time. We claim that this contribution is significant since step (i) is basically a search procedure that is handled by binary search on the path; however in case that a faster procedure is invented, then it can be incorporated in the construction giving the desirable $O(\log\log|U|/\log\log\log|U|)$ bound.

The handling of step (iii) in $O(\log\log|U|/\log\log\log|U|)$ time is a relatively easy task; since the number of elements in the leaf structures is $O(\log|U|)$, we can use for their implementation mechanisms from the *fusion trees* literature [9,10,16]. The main tool we will use is the *q-heap*. The *q-heap* has the following characteristic. *Let Q be a subset of cardinality $|Q| < \sqrt[5]{\log N}$ of a larger set of size N. Then we can store Q in a q-heap data structure of* $O(|Q|)$ *space, such that insertion, deletion, and predecessor queries can be answered in* $O(1)$ *worst-case time. It is assumed that, access is provided to a precomputed table of size* $o(N)$ *that can be constructed in* $O(N^c)$ *time, for some small constant* $c < 1$.

So, consider first the case where the leaf structure has less than $t = (\log|U|)^{1/\log\log\log|U|}$ elements. In this case we represent the leaf structure as a simple balanced tree [11]; this structure permits the handling of insertion, deletion, predecessor, union and split operations in $O(\log t) = O(\log\log|U|/\log\log\log|U|)$ time. Let us now assume that the leaf structure has $s > t$ elements. We represent each leaf structure as an (a, b) tree with $b = 2a - 1$ and $a = \sqrt[5]{\log s}$. The leaves of this tree store the elements of the leaf structure, while the internal nodes store only routing values and are implemented as *q-heaps* (with $N = s$). Hence, we can insert, delete and search in an internal node in worst-case constant time. In order to be also able to split/fuse internal nodes in worst-case constant time we proceed as follows (see also [7]): we represent each internal node u by a pair of nodes u_1, u_2 such that u_1 spans the leftmost a children of u and u_2 spans the remaining at most $a - 1$ children. When a new child is to be added in a node w we check if w is part of a pair. If it is not we create a new node w' that together with w make a pair and move the rightmost child of w to w'. If w is part of a pair we check if it is the left node of the pair. If it is we move the

rightmost child of w to w'; otherwise we simply insert it to w. If both nodes of the pair have now a children we split the pair. Completely symmetrically we can handle the fuse/share operation. So, splitting/fusing/sharing internal nodes corresponds to a constant number of insertion/deletion operations on *q-heaps* and they can be performed in worst case constant time. From the above description it follows that, by implementing insertion, deletion, predecessor and union operations in the usual way for (a, b) trees [11], we get a worst case time cost of $O(h)$ per operation, where h is the height of the (a, b) tree. Since a leaf structure can accommodate up to $O(\log |U|)$ elements we have $h = O(\log \log |U| / \log a) = O(\log \log |U| / \log \log \log |U|)$. Let us now consider the split operation. If we use the usual split operation of the (a, b) tree we can not guarantee an $O(h)$ bound, and this because the splitting of internal nodes during this operation cannot be controlled. We choose, instead of implementing directly this operation, to perform it lazily. In particular, let A be the leaf structure that is going to be split to A_1, A_2. Assume that the number of elements that will be stored in A_1 is k and in A_2 is l. We perform the split operation as follows: we let initially A_1, A_2 be empty and for the next $\min\{k, l\}$ update operations we update suitably either A_1 or A_2 and simultaneously we transfer the leftmost element of A to A_1 and the rightmost element of A to A_2; during these update operations any search operation is answered by querying A_1, A and A_2. It is clear that after at most $\min\{k, l\}$ operations either A_1 or A_2 will have their construction completed and till this time instant any update and query operation will have been performed correctly with a time slowdown of only a constant multiplicative factor. Suppose that A_1 has its construction completed. Then we finish the operation by uniting A and A_2 (in $O(h)$ time) and creating A_2.

Let us now consider step (ii). Suppose that a leaf structure A is split and all the elements of A go to A_1, leaving A_2 empty. The handling of this case will be the main theme of the next sections.

4. The introduction of empty leaves

Suppose that the leaf structure A, which corresponds to an internal node v, is split and one of the two resultant leaf structures is empty. Then the node w of the trie that corresponds to the empty structure becomes an end-node. In that case we call w *empty leaf*. Symmetrically, every end-node pointing at a non-empty leaf structure is called a *filled leaf*. The existence of these two kinds of nodes means that phase 1 of the procedures *Insert*(), *Delete*() and *Pred*() is not complete and has to be further refined. The end-node of the path could now be an empty leaf, or a filled leaf. If it is a filled leaf, nothing changes. But what happens if that node is an empty leaf? Then we have to follow the doubly linked list in order to find the closest to the left filled leaf. The largest distance between two consecutive filled leaves occurs when:

(i) The two paths leading to the filled leaves have only the root of the trie as a common node.
(ii) Both paths have filled leaves only at the lowest level.
(iii) In all the other levels the nodes of these paths are fathers of empty leaves.

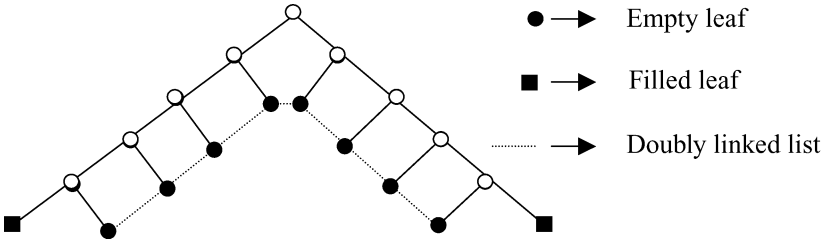

Fig. 2.

Fig. 2 makes clear that, in this case, if we follow the doubly linked list then we have to visit at most $2(\log|U| - \log\log|U|)$ nodes in order to reach a filled leaf. This time complexity is prohibitively large and so we must devise a different data structuring mechanism that could permit the location of the closest filled leaf in (hopefully) $O(\log\log|U|/\log\log\log|U|)$ time.

In order to achieve this we model the previous problem as a *union-split-find* problem [12]. In the union-split-find problem we maintain (initially singleton) sets of elements under the update operations of *union* and *split* and we want to answer queries asking, for a given query element, to locate the set where it belongs (*find* operation). If we treat the empty leaves as elements of sets and consider each filled leaf as the name of the set containing all the empty leaves between this leaf and its nearest (to the right) filled leaf, then the equivalence between the two problems is clear. Moreover the careful reader can notice that in our version of the problem the maximum size that a set can have is $2(\log|U| - \log\log|U|)$. In the next section we present a union-split-find algorithm (Algorithm 1) that is going to be instrumental in handling efficiently the empty leaves problem. The algorithm is based on Blum's union-find algorithm [6] and helps to *partially* solve the aforementioned problem; we use the term partially since some cases remain unresolved. These unresolved cases are handled by our final algorithm (Algorithm 2) that is presented in Section 6.

5. Algorithm 1 for handling empty leaves

5.1. Introduction to the problem and intuition

Consider a universe of n elements $\{0, \ldots, n-1\}$. Initially, every element is a separate set and the name of each set is the name of the element it contains. We want to support the following operations:

- *Union*(x, y): Join the sets x and y $(x < y)$ and produce a new set named x (if a set contains more than one element, then the name of the set is its minimum element). The two sets are sets of consecutive elements and they are adjacent.
- *Find*(x). Find the set that contains element x.
- *Split*(x, y). Let x be the set that contains the element y. Split the set x into sets x and y $(x < y)$. All the elements less than y go to set x and the other elements go to set y.

The proposed algorithm is based on the following data structure:

Definition 1. We define as *L_Tree*, an ordered tree with the following characteristics:

- All the leaves have the same depth.
- The number of the children of all internal nodes (except the root) is between b and $2b - 1$.
- The root has between 2 and $2b - 1$ children.
- Every leaf has a counter. If the counter of the leaf is greater than zero, then that leaf is the name of a set and is called separation leaf; otherwise the leaf simply corresponds to a stored element.
- In each node u, there is a pointer (*min_pointer*) to the minimum element and a pointer (*max_pointer*) to the maximum element stored in the subtree rooted at u.
- Each node u has a pointer to a sorted doubly linked list ($LS(u)$) which contains the children of u ordered according to their left-to-right order.
- Each node u has a pointer to a sorted doubly linked list ($L(u)$) which contains the minimum and the maximum separation leaves from every child of u, ordered according to their left-to-right order.
- Each node u of the tree has a flag (*edge_flag*). When that flag is set to 1, it indicates that u belongs to the leftmost or to the rightmost path of the tree.

The degree of a node u is denoted by $|u|$. Let t be the maximum number of elements that an *L_Tree* can store. Then the maximum height of an *L_Tree* is $O(\log t / \log b)$. The nodes of the tree are partitioned into *levels*. The leaves are at level 0, and the level of a node is equal to the level of its children plus 1. Hence the level of the root is equal to the height of the tree.

We will use *L_Trees* to store (at their leaves) the elements of one (or more) sets in our collection of sets. The basic idea behind the maintenance of the sets is the following. Initially each *L_Tree* corresponds to only one set. Consider an *L_Tree*, which corresponds to a set x. The occurrences of a separation leaf in the $L()$ lists of its ancestors creates a *virtual line* beginning from the separation leaf and ending at its last (highest) occurrence in an $L()$ list. So, in the tree, there exists a virtual line, which starts from the leftmost leaf of the tree (x) and ends at the root of the tree. We call this line *virtual* because it is not maintained by any kind of pointers. If the counter of a leaf z becomes greater than 0 (that is z becomes a filled leaf), we draw a virtual line from z towards the root of the tree. If the counter of a leaf z becomes 0, we delete the virtual line that starts from x. According to the *L_Tree* definition the virtual lines are drawn following the rule that the maximum number of virtual lines that pass from a node u to its parent is 2; the one that corresponds to the minimum separation leaf and the one that corresponds to the maximum separation leaf of the subtree rooted at u.

The operation $Find(x)$ is performed as follows. We start from the leaf x and walk upwards the tree, checking the $L()$ list of each visited node until we find a virtual line that comes from a leaf y, smaller than y. Then, y is the name of the set that contains x.

Suppose now that we want to perform the operation $x = Union(x, y)$. There are two possible cases:

(i) The sets x and y are parts of the same *L_Tree* and are separated by the virtual line that corresponds to element y. In this case we just erase the virtual line that starts from y.
(ii) The sets x and y belong to different *L_Trees*. Let T_x be the *L_Tree* that contains element x, and T_y be the *L_Tree* that contains the element y, and let $x < y$. We erase the virtual line of y and join the two trees.

We are now ready to describe the operations in more detail.

5.2. Detailed description

Split(x, y): To perform the operation *Split*(x, y), we start from the leaf y and walk upwards the tree. In every node u we reach, we access the minimum (*first*$(L(u))$) and the maximum element (*last*$(L(u))$ of $L(u)$). If node u has k ($b \leqslant k \leqslant 2b-1$) children, then the maximum number of elements of this list is $2k$. If *first*$(L(u)) =$ *last*$(L(u)) =$ *nil*, we insert y into the empty list. We set: *first*$(L(u)) = y$, *next*(*first*$(L(u))) =$ *last*$(L(u)) = y$. Let w be the first accessed node for which the list $L(w)$ is not empty. Let $y_1 =$ *first*$(L(w))$ and $y_2 =$ *last*$(L(w))$. If $y_1 < y < y_2$ then we insert y into the list twice and the operation *Split*(x, y) is complete. This costs $O(b)$, which is the size of the list. We do not need to continue walking upwards because y is not going to be inserted in any other list. If $y < y_1$ we insert y into the list twice. This costs only $O(1)$ time because y is to be inserted in the first position. Then we continue upwards. Let z be the node we access. We scan $L(z)$ from left to right until we find the first occurrence of element y_1 and we replace y_1 by y. If y_1 was the leftmost element then the cost of the scan operation is $O(1)$ and we proceed to the above level; otherwise y can not appear in any other list and the algorithm stops, paying a total cost of $O(b)$. Finally the case $y > y_2$ is similar to the previous case and is handled completely symmetrically.

From the above discussion we get the following lemma:

Lemma 1. *The time complexity of the Split() operation is* $O(b+h)$ *where b is the branching factor and h the height of the L_Tree.*

Find(x): We start from x and walk upwards the tree. For every node u we reach, we access the first and the last element of its list. If $x >$ *last*$(L(u))$ then *last*$(L(u))$ is the name of the set that contains x. If $x <$ *first*$(L(u))$ we continue upwards. If *first*$(L(u)) < x <$ *last*$(L(u))$, we find the maximum element of the list $L(u)$ which is smaller than x. This element is the name of the set that contains x.

Lemma 2. *By executing the operation Find*(x)*, we find the name of the set that contains* x.

Proof. Let y be the name of the set that contains x and suppose that *Find*(x) did not return y, but another element, z. Then, according to the way the lists $L()$ have been built we should have: $x > z > y$. But since z is a separation leaf, then x belongs to the set z, which is not valid. □

From the description of the *Find*() procedure the next lemma follows:

Lemma 3. *The complexity of the Find() operation is* $O(b+h)$ *where b is the branching factor and h is the height of the L_Tree.*

Union(q, s): If the sets q, s belong to the same *L_Tree*, then we start from s and walk upwards the tree. For every node we reach we delete s from its $L()$ list. In fact we perform the inverse procedure of *Split*(s). The time cost for the *Union*() in this case is $O(b+h)$.

Assume now that the sets q and s belong to the *L_Trees*, T_A (with height A) and T_B (with height B) respectively. We assume w.l.o.g. that $A > B$. Firstly, we delete s from all the $L()$ lists it appears. In order to implement efficiently the *Union*() operation we have equipped every node u of an *L_Tree* that has $g > b$ children and belongs to the rightmost (leftmost) path of the tree, with a *candidate_right* (*candidate_left*) node, which is also the father for the g-b rightmost (leftmost) children of u. So, every node has two possible fathers which are accessible via pointers ($father_1, father_2$). If w is the *candidate_right* (*candidate_left*) node of u, then for each of the g-b rightmost (leftmost) children of u, we have stored $father_1 = u$ and $father_2 = w$. Let r be one of these children. In order to find the father of r, we access $father_1$ and $father_2$. If both are valid, let x, y be the minimum and the maximum leaves of the subtree rooted at u; and z, f be the minimum and maximum leaves of the subtree rooted at w. Then w is the father of r if $z \leqslant x < y \leqslant f$. The introduction of the *candidate_right* and *candidate_left* nodes allow constant time splitting of nodes and permit every level (except from the first and the last) to be processed in $O(1)$ time by the union algorithm. We can assume that the candidate nodes do not exist. A candidate node becomes a valid node of the tree when the number of its children becomes at least b.

We can now proceed in the detailed description of the operation *Union*(q, s). We locate the rightmost node (u) of T_A with height B. Let w be the root of T_B. After locating w, we execute procedure *Union_Level*$(u, LS(w))$.

Procedure Union_Level$(u, LS(w))$

Case 1: The list $LS(w)$ has size $g \geqslant b$.
If u has a *candidate_right* node, then we delete that node.
We create a new node $z =$ *candidate_right*(w). /* w is considered a global variable */
We cross the list $LS(w)$ from left to right.
For every node y that belongs to the first b accessed nodes we set:

$$father_1(y) = w$$
$$father_2(y) = nil$$

For the remaining nodes we set:

$$father_1(y) = w$$
$$father_2(y) = z$$

Let x be the first of these remaining nodes. We store in z a pointer to x (*LS_cut_pointer*).

We cross the list $L(w)$ from left to right until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at x.
We store in z a pointer to that element (*L_cut_pointer*).
Let r be the father of u. (If r does not exist we create a new node r with child u.)
Union_Level$(r, \{w\})$.

Case 2: The list $LS(w)$ has size $g \leqslant b$. We have the following scenarios:

2(a) $|LS(u)| + |LS(w)| < 2b$. If $|LS(u)| = b$, then u does not have a *candidate_right* node. In that case we create a new node $z =$ *candidate_right*(u).
We store in z a pointer (*LS_cut_pointer*) to *first*$(LS(w))$ and a pointer (*L_cut_pointer*) to *first*$(L(w))$. If $|u| > b$ then z already exists.
We cross the list $LS(w)$ from left to right and for every node y we set:

$$father_1(y) = u$$
$$father_2(y) = z$$

We connect $LS(w)$ at the end of $LS(u)$. The new list is $LS(u)$.
We connect $L(w)$ at the end of $L(u)$. The new list is $L(u)$.
We update the *min-max* pointers for node u.
Return.

2(b) $|LS(u)| + |LS(w)| = 2b$. Let $z =$ *candidate_right*(u).
Node z is no longer the *candidate_right* node of u, and becomes a valid node of the tree.
Using the pointers stored in z (*LS_cut_pointer*, *L_cut_pointer*), we split the lists $LS(u)$, $L(u)$. The left parts are the new lists $LS(u)$, $L(u)$ and the right parts are the lists $LS(z)$, $L(z)$.
We update the *min-max* pointers of u.
We walk the list $LS(w)$ and for every node y we access, we set $father_1(y) = z$.
We connect $LS(w)$ at the end of $LS(z)$. The new list is $LS(z)$.
We connect $L(w)$ at the end of $L(z)$. The new list is $L(z)$.
We store in z the *min-max* pointers according to the children of z.
Let r be the father of u. (If r does not exist we create a new node r with child u.)
Union_Level$(r, \{z\})$.

2(c) $|LS(u)| + |LS(w)| > 2b$. Let $z =$ *candidate_right*(u).
Node z is no longer the *candidate_right* node of u, and becomes a valid node of the tree.
Using the pointers stored in z, we split the lists $LS(u)$, $L(u)$. The left parts are the new lists $LS(u)$, $L(u)$ and the right parts are the lists $LS(z)$, $L(z)$.
We update the *min-max* pointers of u.
We set $w =$ *candidate_right*(z).
We cross $LS(w)$ from left to right and for every node y we access we store $father_1(y) = z$.
We connect $LS(w)$ at the end of $LS(z)$. The new list is $LS(z)$.
We connect $L(w)$ at the end of $L(z)$. The new list is $L(z)$.
We cross the first b nodes of $LS(z)$ from left to right.

For each node y that belongs to the remaining nodes of $LS(z)$ we set:

$$father_1(y) = z$$
$$father_2(y) = w$$

Let p be the first of these remaining nodes. We store in w a pointer to p (*LS_cut_pointer*).
We cross the list $L(z)$ from left to right until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at p.
We store in w a pointer to that element (*L_cut_pointer*).
We update the *min-max* pointers of w.
Let r be the father of u. (If r does not exist we create a new node r with child u.)
Union_Level$(r, \{z\})$.

Lemma 4. *After joining the trees T_A, T_B, the sets q and s have been joined.*

Proof. q is the rightmost separation leaf of T_A. Suppose we start walking upwards the tree from a leaf t that belonged to set s before the deletion of s from all the lists of T_B. The first virtual line we are going to find is the one that corresponds to separation leaf q because q is the maximum separation leaf of the new tree, which is smaller than t. Hence, after joining T_A and T_B, the sets q, s have been joined. □

Lemma 5. *The time complexity of the Union() operation is $O(b+h)$ where b is the branching factor and h is the largest height of the L_Trees involved in the union.*

Proof. Let h_1, h_2 be the heights of the two trees involved in the union operation with $h = h_1$. After locating the root of the tree with the smaller height (time cost $O(h_2)$), we can divide the union operation into three phases:

Phase 1 consists of the first execution of *Union_Level*. The time cost for phase 1 is $O(b)$ because this is the size of the list $LS(w)$. Also, in case 2(c), part of the list $LS(u)$ must be scanned. The size of that list is also $O(b)$. All the other updates cost $O(1)$.

Phase 2 consists of all the other executions of *Union_Level* except from the last two. Each execution of *Union_Level* in this phase costs $O(1)$. The list $LS(w)$ is replaced by a single node. Furthermore, case 2(c), will never occur, because after the addition of at most one child, the candidate node will have at most b children. The time cost of this phase is $O(h - h_2)$.

Phase 3 consists of the two final executions of *Union_Level*. The final one (case 2(a)) costs $O(1)$ time, while the last before the final one costs at most $O(b)$.

Therefore the total time cost is $O(h - h_2) + O(h_2) + O(b) = O(h + b)$. □

Algorithm 1 is based on the union-find algorithm of Blum. The main difference between Algorithm 1 and Blum's algorithm is that every node of the tree (and not only the root) has between b and $2b - 1$ children. Algorithm 1 is not enough for the manipulation of the

empty leaves since there are some unresolved cases. In the next section we will see the problems that remain unsolved, and we shall describe Algorithm 2, that finally solves the problem.

6. Algorithm 2 for handling empty leaves

6.1. Required characteristics

Although Algorithm 1 is an important step towards the solution of the empty leaves problem there are some problems that still remain unsolved. Suppose that the empty leaves $u_1, \ldots, u_8$ (Fig. 3) are stored at the leaves of an *L_Tree*. Assume now that we start inserting elements into u_6. When the number of elements of the leaf structure of u_6 becomes $\log\log|U| + i + 1$ (i is the depth of u_6), then the leaf structure of u_6 is split into two parts (*Open*()). What happens if one of these parts (w, for example) is empty? We have to create an *L_Tree* containing w, u_7 and u_8. Hence, by the time u_6 is split, u_7 and u_8 must form an *L_Tree*. This cannot be achieved so far, because Algorithm 1 does not produce separate trees as a result of a *Split*() operation. We need an algorithm that *produces separate trees within a certain number of steps*, that is we seek for an implementation of the *Split*() operation that: proceeds incrementally; it is completed in at most $O(\log\log|U|)$ incremental steps (this is the minimum size of a leaf structure before it splits); it can undo its incremental operations during removal of elements.

In particular, every time we insert an element into the leaf structure of an end-node stored at a leaf of an *L_Tree*, we must perform an incremental separation step for that leaf. Also, every time we delete an element from a leaf structure of an end-node stored at a leaf that belongs to an *L_Tree*, we must undo the last separation step we did for that leaf. The available time for doing or undoing a separation step, is $O(\log\log|U|/\log\log\log|U|)$. So, we have to reach Target 1 for Algorithm 2.

Target 1. *The result of performing* $\log\log|U|$ *separation steps for a leaf of a tree, is that the leaf belongs to an L_Tree with only one element.*

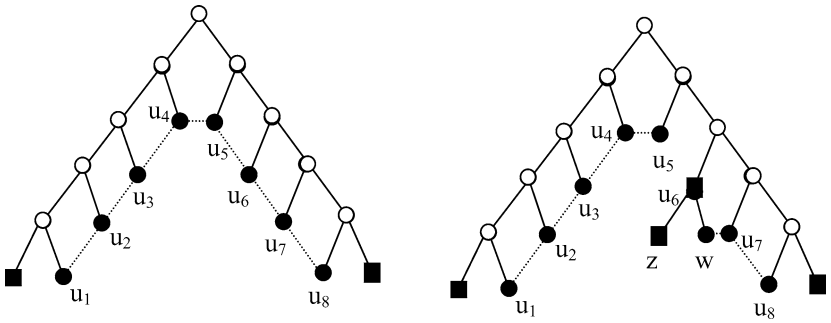

Fig. 3.

Actually, Algorithm 2 achieves that, after $f = O(\log\log|U|/\log\log\log|U|)$ separation steps.

Suppose now that a *Close*() operation is performed for w and its brother, z. What happens if w or z belonged to an *L_Tree*? If w belonged to an *L_Tree*, then z did not belong to an *L_Tree*, because its leaf structure had more than $2f$ elements. Hence, w was the leftmost leaf of the *L_Tree*. When *close*() occurs, leaf w stops to exist. This means that w has to be separated from the tree it belongs to. So we have to reach Target 2.

Target 2. *If u is the rightmost leaf or the leftmost leaf of an L_Tree, we can produce a proper L_ Tree containing all elements except u in* $O(\log\log|U|/\log\log\log|U|)$ *time.*

In order to achieve Targets 1, 2 we should devise a split procedure that creates separates trees; unfortunately a straightforward implementation of such procedure could cost $O(bh)$ time which is unacceptably high. So we chose to reimplement Algorithm 1 by devising a novel split procedure that *incrementally prepares* the creation of separate trees; the careful reader should note that the straightforward algorithm could possibly get an acceptable amortized bound, however we seek for worst case time bounds and our proposed incremental algorithm can be considered as a mechanism to convert the amortized time bound into a worst-case time bound. The successful implementation of this incremental split procedure implies the proper modification of the other operations. So, in Section 6.2 we describe the incremental steps that have to be performed at *every* insertion/deletion of an element in a leaf structure; in Section 6.3 we describe the *Split*() operation that is performed when a leaf becomes filled, that is when the respective leaf structure gets its *first* element; and in Section 6.3 we describe the union operation. We should note that in our new algorithm a complete execution of the split procedure entails an execution of the *Split*() procedure plus an execution of at most $\log\log|U|$ incremental separation steps; the *Find*() operation remains unchanged. However it is possible, as a by-product of our construction, to create *L_Trees* that do not contain any separation leaf. This event can possibly spoil the time complexity of our algorithm, if long chains of such *L_Trees* are created. In Section 7 we show that this is not the case; in particular we prove that the number of consecutive *L_Trees* that can be created is at most 2 and so the time complexity of the procedure remains reasonably small.

Finally we should note that in this new algorithm each *L_Tree* contains at most $O(\log|U|)$ elements since the completion of a split operation results to the production of separate *L_Trees*, before an *Open*() operation is performed (see also Fig. 2). So, by setting $b = O(\log\log|U|/\log\log\log|U|)$, we get that the maximum height of an *L_Tree* during the course of the algorithm is $O(\log\log|U|/\log\log\log|U|)$. Till the end of the paper H will be used to denote this maximum height.

Now we are ready to proceed to the description of Algorithm 2, which has the characteristics mentioned above.

6.2. The leaf separation steps

In every node of an *L_Tree* there is a counter. The counter of a leaf is equal to the number of separation steps that have been performed on the leaf. The value of the counter

of an internal node is the number of the separation leaves in its subtree, for which the incremental separation steps have reached levels above the level of the internal node. So internal nodes with positive counter are nodes that have been affected by an incremental operation.

There are two kinds of nodes:

- *Destination nodes.* The children of a destination node may have counters with value greater than 0. If at least one child of a destination node has a positive counter, then the leftmost and the rightmost child of that node, have positive counters.
- *Pool nodes.* The counters of all the children of a pool node are equal to zero.

The intuitive explanation of the role of these two kinds of nodes is the following: a node of Algorithm 1 is equivalent to a configuration of two pool nodes and one internal node of Algorithm 2; if a split is completed then the two pool nodes will be nodes of separate trees; a split is undone by uniting a configuration of two pool nodes and a destination node to a single destination node.

Initially all nodes in the tree are destination nodes.

During the functioning of the algorithm, each leaf can be in one of two phases:

- *Bottom-up phase:* We say that a leaf of the tree is in bottom-up phase, if the value of its counter is less than the level of the root of the tree.
- *Top-down phase:* We say that a leaf of a tree is in top-down phase, if the value of its counter is equal or greater than the level of the root of the tree.

Intuitively a leaf is in the bottom-up phase when the separation process has already reached the root (effectively separating the tree) and some additional update operations are required on the separated trees. Also one forward separation step for a leaf v, at least for the bottom-up phase consists of exactly all necessary updates at the next internal node on the path from v towards the root, while the backward step undoes the last forward step.

We are now ready to proceed to the detailed description of the separation steps.

6.2.1. Bottom-up phase

6.2.1.1. Forward steps. Suppose that the counter of a leaf w is increased from $i-1$ to i. If $i = 1$, we split the set where w belongs, at w (this split process is similar to that in Algorithm 1 and it will be described later). If $i > 1$ we ascend to the ancestor of w at level $i-1$ (let this be w_j) and increase its counter by 1. If the counter is already positive, nothing happens. If the counter was 0, we distinguish the following cases:

Case 1: The father of w_j is a destination node u, whose all other children have counters equal to 0. We create the pool nodes u_L and u_R. u_L is called *left pool* of the destination node u and u_R is called *right pool* of the destination node u. All the siblings of w_j that lay right of w_j move to u_R, and all the siblings of w_j that lay left of w_j move to u_L. Finally w_j becomes the only child of u. Fig. 4 depicts the aforementioned operations.

Actions for future use. The actions described below may not make sense at this point. The purpose of these actions will become clear during the description of the *Union*() algorithm, later on. Roughly, the purpose of these actions is the following. u_L after the

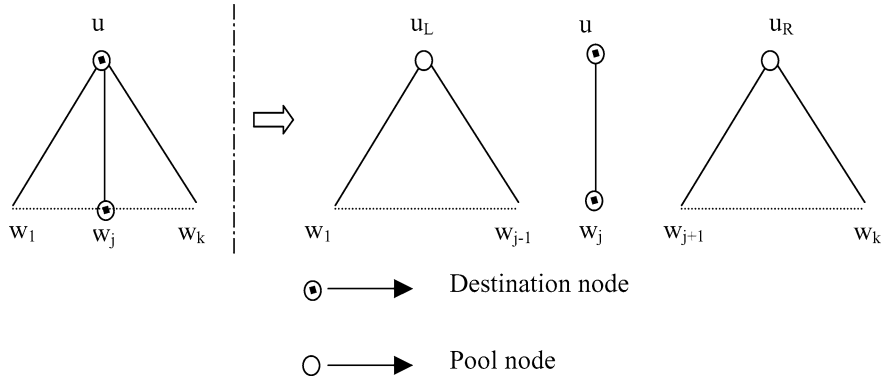

Fig. 4.

completion of the incremental operations may become node of the rightmost path of an *L_Tree*. Symmetrically, u_R may become node of the leftmost path of an *L_Tree*. In Algorithm 1 we mentioned that every node that belongs to the leftmost (rightmost) path of a tree and has more than b children, has a *candidate_left* (*candidate_right*) node. Furthermore, after the completion of the incremental operations, u_L, u_R may have less than b children which does not comply with the definition of the *L_Tree*. So, special actions must be performed in order to make these nodes ready to be treated by the tree separation step, which will be presented later on (see Section 6.4.1). The purpose of these actions is to "fix" the nodes of the leftmost or the rightmost path of a tree, so that they have between b and $2b-1$ children.

Definition 2. If a node has between b and $2b-1$ children is called a heavy node.

We are now going to describe the necessary actions for the handling of u_L. The actions for u_R are symmetrical. We distinguish the following scenarios:

(i) If $|u_L| \geqslant b$ then u_L will be heavy when it becomes a node of the rightmost path of a tree. All we need to do is to create a new node $w = \mathit{candidate\_right}(u_L)$ according to Algorithm 1.

(ii) Assume now that $|u_L| < b$.
If the node on the left of u_L (let z be that node) is a pool node, or a destination node, which has children with positive counters, then nothing has to be done.
If node z is a destination node having no child with positive counter, then we distinguish the following cases:
  (i) $|z| + |u_L| < 2b$.
  We cross the list $LS(z)$ from left to right.
  For every node y that belongs to the first b accessed nodes we set:

$$\mathit{father}_1(y) = z$$
$$\mathit{father}_2(y) = \mathit{nil}$$

For the remaining nodes we set:

$$father_1(y) = z$$
$$father_2(y) = u_L$$

Let r be the first of these remaining nodes. We store in u_L a pointer to r (*LS_cut_pointer*).
We scan the list $L(z)$ from left to right, until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
We store in u_R a pointer to that element (*L_cut_pointer*).

(ii) $|z| + |u_L| \geqslant 2b$. (Let $|u_L| = b - x$.)
We scan the list $LS(z)$ from right to left.
For every node y that belongs to the first x accessed nodes, we set:

$$father_1(y) = z$$
$$father_2(y) = u_L$$

Let r be the first of the last x nodes. We store in u_L a pointer to r (*LS_cut_pointer*).
We cross the list $L(z)$ from left to right until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
We store in u_L a pointer to that element (*L_cut_pointer*).

Case 2: The father of w_j is a pool node; let this be a left pool node u_L (completely symmetrically it could be a right pool u_R). Let u be the destination node from which u_L was formed. Node w_j and all the children of u_L that lay right of w_j, are moved to u (see Fig. 5).

If w_j is a child of u_R (the right pool of u), the actions are symmetrical.

Actions for future use. The actions for the treatment of u_L (or u_R in the symmetrical case) are the same as in case 1.

Case 3: The father of w_j is a destination node u, having children with counters different from 0. Hence, according to the definition, the leftmost and the rightmost child of u, have positive counters (Fig. 6). In this case we do nothing.

6.2.1.2. Backward steps. Suppose that the counter of a leaf w is decreased from i to $i - 1$. If $i = 1$, we perform the inverse procedure of *Split*() (*Split*() will be described later

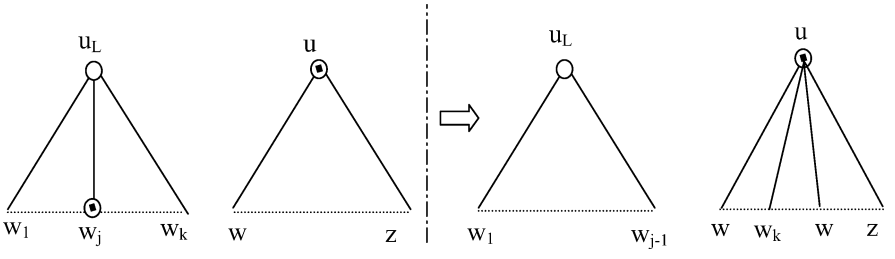

Fig. 5.

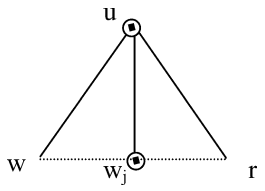

Fig. 6.

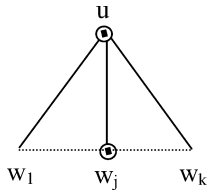

Fig. 7.

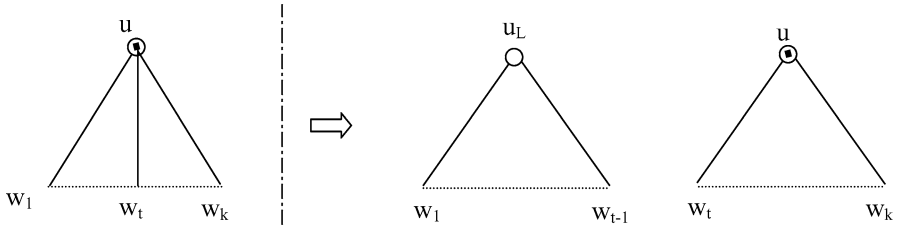

Fig. 8.

on). If $i > 1$ we ascend to the ancestor of w at level $i - 1$ let this be w_j and decrease its counter by 1. If the counter is still positive, nothing happens. If the counter becomes 0, we distinguish the following cases:

Case 1: The father of w_j is a destination node u; the counters of the leftmost and rightmost child u are positive; and w_j is neither the leftmost nor the rightmost child of u (Fig. 7). In this case we do nothing.

Case 2: The father of w_j is a destination node u; the counters of the leftmost and rightmost children of u are positive; and w_j is the leftmost child w_1 of u (the case where w_j is the rightmost child w_k of u is completely symmetrical). In this case let w_t be the closest to w_1 child of u with positive counter. We just have to make $w_1, \ldots, w_{t-1}$ become children of the left pool of u, which is created in case it does not exist. Fig. 8 depicts the aforementioned actions.

If instead of the counter of w_1, the counter of w_k becomes zero, the case is actually the same (the actions are symmetrical).

Actions for future use. The actions for handling u_L (or u_R in the symmetrical case) are the same as in case 1 at the forward steps.

Case 3: The father of w_j is a destination node u; and w_j is the only child of u.

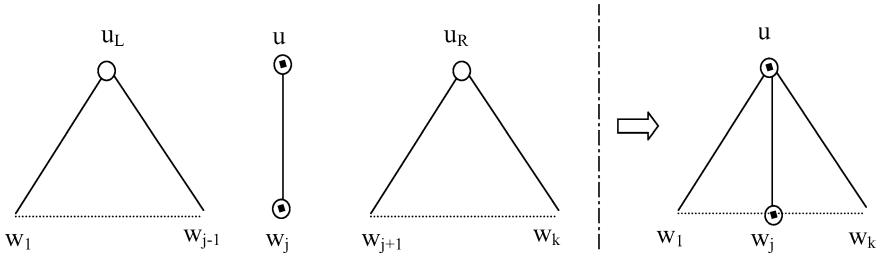

Fig. 9.

In this case, we just have to reunite the configuration of the two pool nodes and the destination node, to a single destination node u. Fig. 9 depicts graphically the result of undoing the separation step.

Actions for future use: We check if u belongs to the leftmost or to the rightmost path of the tree, or if the node next to u (on the left side or the right side) belongs to the leftmost or to the rightmost path of the tree. We distinguish the following scenarios:

(i) u belongs to the rightmost path of the tree. Then
   1. If $|u| \geqslant b$, we create a node $z = candidate\_right(u)$ according to Algorithm 1.
   2. If $|u| < b$ ($|u| = b - x$), we go to the node next to u (on the left side of u). Let z be that node. If z is a destination node, which has children with positive counters, or z is a pool node, we do nothing. Otherwise:
      1. If $|u| + |z| \geqslant 2b$ then
         We cross the list $LS(z)$ from right to left.
         For every node y that belongs to the first x accessed nodes we store:

$$father_1(y) = z$$
$$father_2(y) = u$$

         Let r be the first of the last x nodes. We store in u_L a pointer to r (*LS_cut_pointer*).
         We scan the list $L(z)$ from left to right, until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
         We store in u a pointer to that element (*L_cut_pointer*).
      2. If $|u| + |z| < 2b$ then
         We scan the list $LS(z)$ from left to right.
         For every node y that belongs to the first b accessed nodes we set:

$$father_1(y) = z$$
$$father_2(y) = nil$$

         For the remaining nodes we set:

$$father_1(y) = z$$
$$father_2(y) = u$$

Let r be the first of these remaining nodes. We store in u a pointer to r (*LS_cut_pointer*).
We scan the list $L(z)$ from left to right until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
We store in u a pointer to that element (*L_cut_pointer*).

(ii) The node on the right side of u belongs to the rightmost path of the tree. Let z be that node. If z is a destination node which has children with positive counters, then we do nothing. Otherwise, z is a destination node whose children have zero counters. We distinguish the following cases:

1. If $|z| + |u| < 2b$ then
   We cross the list $LS(u)$ from left to right.
   For every node y that belongs to the first b accessed nodes we store:

   $$father_1(y) = u$$
   $$father_2(y) = nil$$

   For the remaining nodes we store

   $$father_1(y) = u$$
   $$father_2(y) = z$$

   Let r be the first of these remaining nodes. We store in z a pointer to r (*LS_cut_pointer*).
   We scan the list $L(u)$ from left to right until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
   We store in z a pointer to that element (*L_cut_pointer*).
2. If $|u| + |z| \geqslant 2b$ then
   We scan the list $LS(u)$ from right to left.
   For every node y that belongs to the first x accessed nodes we store:

   $$father_1(y) = u$$
   $$father_2(y) = z$$

   Let r be the first of last of the x nodes. We store in z a pointer to r (*LS_cut_pointer*).
   We cross the list $L(u)$ from left to right, until we reach the minimum element of the list that belongs to a subtree on the right side of the subtree rooted at r.
   We store in z a pointer to that element (*L_cut_pointer*).

(iii) Node u belongs to the leftmost path of the tree. In this scenario the actions are symmetrical with the actions of scenario 1.

(iv) The node on the left side of u belongs to the leftmost path of the tree. In this scenario the actions are symmetrical with the actions of scenario 2.

Remark. A forward or backward bottom-up separation step performed at a level of a tree may affect the above levels. A step (forward or backward) may be considered as a transaction between a pool node and the destination node. This will affect all their ancestors until their nearest common ancestor. For all these nodes the *min-max* pointers must be updated.

Also, their $L()$ lists must be updated. This update costs O(1) for every node because only the first or the last element of the list is to be accessed.

Lemma 6. *The time complexity of a bottom-up* (*forward or backward*) *separation step is* $O(b + H)$.

Proof. Locating the node that is to be modified by the separation step costs $O(H)$ time. The father of $O(b)$ nodes will change. The actions for future use also cost $O(b)$. If the above levels are affected, the changes cost O(1) for each level. Therefore, the time complexity of a separation step is $O(b + H)$. □

6.2.2. Top-down phase

6.2.2.1. Forward steps. When the separation steps for a leaf x of the tree reach the root of the tree, x enters the top-down phase. From now and on, every time we increase the counter of x, we shall be doing a top-down separation step for that leaf.

Let u be the root of the tree and w be the child of the root which is an ancestor of x. From the tree with root u, we create at most three new trees (Fig. 10). The root of the first tree (u_1) has as children all the children of u that lay left of w. The root of the second tree is w. The root of the third tree (u_2) has children all the children of u that lay right of w.

When a new tree is created by a top-down separation step, a pool node may be separated from its destination. This means that after the separation, the pool node belongs to a tree A while the destination node belongs to a tree B. This pool node must become a destination node. Hence, for every tree, we must search the leftmost and the rightmost path for pool nodes that have been separated from their respective destination node. Every such node must become a destination node. The cost for this action is $O(H)$. Up to now it seems easy to do this task every time a new tree is created, because the number of the new trees is at most three. Later on, we will see that the effective implementation of the *Union* operation requires the creation of up to H new trees at the same time, thus making the total time cost prohibitely large. Hence, we partition the procedure implementing this task into incremental pieces (termed *undone jobs*) that are executed at each separation step and that have $O(H)$ time cost.

Definition 3. We call dept list of a tree, a list that contains all the leaves of the tree (with more than one leaf) which have entered the top down phase and at least one forward top-down separation step has been performed for them.

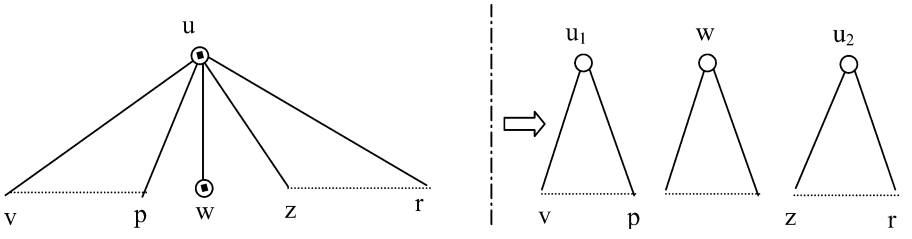

Fig. 10.

Lemma 7. *The maximum size of a dept list is the maximum height of an L_Tree.*

Proof. Suppose we perform a forward top-down separation step for a leaf x, and let m be the number of elements of the *dept list*. If we create a tree that does not contain x, the height of that tree is equal or less than the height of the initial tree. Since x is not contained in the tree, the *dept list* may be, in the worst case, the *dept list* of the initial tree. If the new tree contains x, its height is reduced by 1, compared with the height of the initial tree. Its *dept list* may be, in the worst case, the *dept list* of the initial tree plus x, if x was not an element of that list. This means that the size of the *dept list* may be increased by 1 only if the height of the tree is decreased by 1. As a consequence, the maximum size of a *dept list* is equal to the maximum height of a tree. □

Remark. If $L(u_1)$ (or $L(u_2)$) is empty then it is possible that the tree rooted at u_1 (u_2) has no separation leaves. Therefore, a set may be contained in more than one trees. In Section 7 (the forest *of L_Equivalent* trees) we will show how a set can be contained in at most three *L_Equivalent* trees. So, given a leaf *y,* the maximum number of trees we need to search, in order to find the name of the set that contains y (the closest filled leaf in the left direction), is three.

6.2.2.2. Backward steps. A leaf x is in top down phase and its counter is reduced by one. Let C be the new value of the counter and A be the level of the root (leaves lay on level 0). If $C > A - 1$ we do nothing. Otherwise x returns to bottom-up phase, and we perform a bottom-up backward step for x at level $A - 1$. x is deleted from the *dept list* of the tree. The time cost of a top-down backward separation step is 0 when $C > A - 1$. If $C = A - 1$ the cost is $O(b + H)$ but in this case, we actually perform a bottom-up backward separation step.

Lemma 8. *The time complexity of a top down* (*forward or backward*) *separation step is* $O(b + H)$.

Proof. For the forward top-down separation step, the cost for the creation of the three (at most) new trees is $O(b)$. To cut the *dept list* of the initial tree into parts according to the new trees created (every new tree must now have its own *dept list* which is part of the initial list), we have to cross the initial list and the cost for this is (according to Lemma 7) $O(H)$. Therefore, the total cost of the forward top-down separation step is $O(b + H)$. For the backward top-down separation step the maximum cost arises when we have to return to the bottom up phase and perform a bottom-up backward separation step. In this case according to Lemma 6 the cost is $O(b + H)$. □

6.3. Split(x, y)

Split(x, y) is performed when the counter of the leaf y is increased from 0 to 1. The description is similar to that of Algorithm 1, with the addition of some new actions introduced by the bottom-up phase. The difference from Algorithm 1 is that when we insert x into a list $LS(u)$ (and not replace another element by x), we may have to update an

L_cut_pointer. If u belongs to the rightmost (leftmost) path of the tree, we have to update the *L_cut_pointer* of the *candidate_right* (*candidate_left*) node of u. If u lays on the left (right) of a node w that belongs to the rightmost (leftmost) path of the tree, we may have to update *L_cut_pointer*(w), if this pointer is already set in w. To update an *L_cut_pointer* costs O(1), so as in Algorithm 1 we get the following lemma:

Lemma 9. *The total time cost for Split*(x, y) *is* $O(b + H)$.

6.4. *Union*(q, s)

Definition 4. We call a tree with separation leaves, *L_Equivalent*, if it has the following characteristic: if we undo all the separation steps for all the separation leaves of the tree, then all the nodes of the tree will have between b and $2b - 1$ children, except from the nodes that belong to the leftmost or to the rightmost path. These nodes may have less than b children.

The height of an *L_Equivalent* tree is $O(H)$, where H is the maximum height of an *L_Tree* with the same number of leaves.

The top-down phase introduces, for the union operation, Requirement 1.

Requirement 1. A tree with a non-empty dept list cannot take part in a union operation.

This way, the size of the *dept lists* is controlled. Furthermore, if such a union was possible, it could possibly destroy forward top-down separation steps.

Furthermore from the description of the top-down phase, when an *L_Equivalent* tree is split, the created trees are *L_Equivalent*. The problem is that if we join two *L_Equivalent* trees, the resulting tree will not be *L_Equivalent* because internal nodes may have less than b children, if all the separation steps are undone. This observation leads to Requirement 2.

Requirement 2. If a tree is going to be the left part of a union operation then we must process that tree in such a way that if all the separation steps for all the leaves of the tree are undone, then the nodes of the rightmost path of the tree will have between b and $2b - 1$ children. Symmetrically, if a tree is going to be the right part of a union operation then we must process that tree in such a way that if all the separation steps for all the leaves of the tree are undone, then the nodes of the leftmost path of the tree will have between b and $2b - 1$ children.

Lemma 10. *If Requirement* 1 *and Requirement* 2 *are met, Target* 1 *is achieved.*

Proof. The bottom-up separation steps are bounded by the maximum height of an *L_Equivalent* tree which is $O(\log\log|U|/\log\log\log|U|)$. This means that when the counter of a leaf x, reaches a value $f = O(\log\log|U|/\log\log\log|U|)$, x definitely enters the top-down phase. When the first top-down separation step is performed for x, if x is not a root of a tree, it is inserted into the *dept list* of the tree it belongs. As long as the *dept list* of the tree is not empty, the tree is no longer allowed to take part in a *union*

operation. Hence, every time the counter is increased while x is an element of the *dept list*, an ancestor of x becomes root of a tree. When the value of the counter of becomes $2f$, x will become a root. □

We are now ready to proceed to the description of the *Union* operation. If the sets q, s belong to the same *L_Equivalent* tree, we start from s and walk upwards the tree. For every node we reach, we delete s from its list, by performing the inverse procedure of *split*(s).

Suppose now that the set q is included, in an *L_Equivalent* tree T_v with root v and the set s is included in an *L_Equivalent* tree T_y with root y. The trees T_v and T_y may not be ready for the union operation of Algorithm 1. Even if Requirement 1 and Requirement 2 are met, in Algorithm 1 the nodes are not affected by the leaf separation steps. If there are nodes at the rightmost path of T_v, affected by the leaf separation steps, Algorithm 1 does not work. The tree separation steps, described bellow, create from T_v a new *L_Equivalent* tree that contains the set q, or part of the set q. The nodes of the rightmost path of the new tree are not affected by the leaf separation steps, which means that they are destination nodes, which have no children with positive counter. Applied to T_y, the tree separation steps create a new tree that contains the set s (or part of the set s) and all the nodes of its leftmost path are destination nodes, which have no children with positive counter.

For reasons of clarity we will distinguish these tree separation steps into two types that are performed sequentially: *the tree separation step* (is performed first) and the *root separation step* (is performed second and produces the final output).

6.4.1. The tree separation step

We will show how to process T_v. T_y is processed in a symmetrical way. Let u be a node of the rightmost path of T_u. Fig. 11 depicts all the different cases we are going to face in every level of the tree, concerning the status of u.

Let $v, u_1, \ldots, u_k$ be the nodes of the rightmost path of T_u. Let $u_{j_1}, u_{j_2}, \ldots, u_{j_L}$ be the nodes of this path that fall into cases 1, 2, 3 and 4. Every one of these nodes is going to become the root of a tree.

We will present the handling of these nodes in a modular way by describing the procedure *Tree_Extraction*(u_1, u_2, c). Here u_1, u_2 are nodes of the rightmost path, with u_1 being an ancestor of u_2 and c is an integer taking two values 0 or 1. The procedure assumes that all nodes between u_1 and u_2 fall into case 5 and that u_2 falls into cases 1, 2, 3, 4. The third

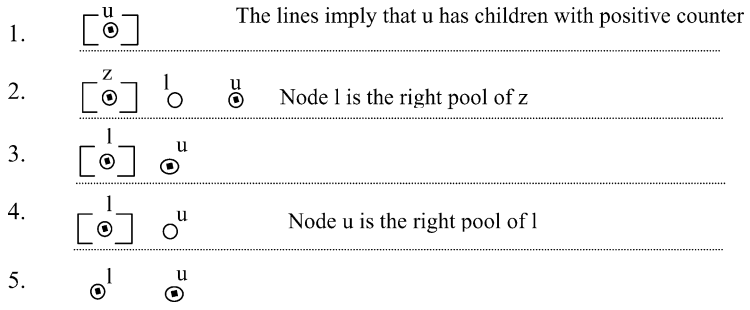

Fig. 11.

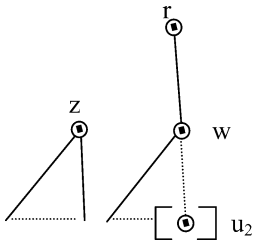

Fig. 12.

argument of the procedure may be one or zero. If it is one, the second argument (u_2) is going to become the root of a separate tree, which is extracted from the tree rooted at the first argument of the procedure; otherwise a separate tree is not produced.

The procedure starts from u_2 and walks upwards the tree "fixing" the rightmost path of the tree, that is making the nodes of the rightmost path heavy.

In more details:

Procedure Tree_Extraction(u_1, u_2, c)

We start from u_2 and walk towards u_1, doing some process in each level. Our goal is to be able to process each level in O(1) time. In order to achieve this goal, we are going to use the information stored during *the actions for future use*, introduced in the bottom-up phase. We are now going to describe this processing.

Let w be the father of u_2. Firstly we have to process the level of w.

Suppose that u_2 falls into cases 1, 3 or 4. We delete u_2 from $LS(w)$ and $\min(L(w))$, $\max(L(w))$ from $L(w)$ (Fig. 12).

Node w definitely falls into case 5. If w has no children after the deletion of u_2, we delete w. Suppose, now that w still has children.

- $|w| \geqslant b$ (but there is no pointer-node pointing to w; we will explain immediately what a pointer-node is)
  - If w was not created by a leaf separation step, it has a candidate node.
  - If w was created by a leaf separation step, we have stored a node $z = \textit{candidate\_right}(w)$, according to Algorithm 1. If during that time z was in the present state, the *candidate_right* node was stored as a result of the actions for future use in case 1, or in case 2 of the forward steps. If z was not in the present state, the candidate node of w was created when we visited z for the last time (see the actions for future use of case 3 for backward steps).
- $|w| < b$ ($|w| = b - x$). If $x = 1$ then before the deletion of u_2, w had b children and maybe it was not created by a leaf separation step. If $|z| > b$ we move the last child of z to w and update the lists $LS(z)$, $L(z)$, $LS(w)$, $L(w)$ and the *min-max* pointers of z, w. If $|z| = b$, we make z a pointer to w, which means that we perform the following actions. We join $LS(z)$ with $LS(w)$. The new list is $LS(w)$. We connect $L(z)$ with $L(w)$. The new list is $L(w)$. Now, w is the common father of all children of z, but from these children, w is accessible only via an intermediate node, z. So we say that z is transformed into a pointer to w. Node w now takes the place of z.

If $x > 1$, the node was definitely created by a leaf separation step. The possible scenarios for this creation are those mentioned below:

- $|w| + |z| \geqslant 2b$. Let p be the child of z such that the number of w's children on the right side of p, is $x - 1$. For p and every one of these children we have set their $father_1$ and $father_2$ pointers to z, w respectively. We have also stored in w a pointer (*LS_cut_pointer*) to p. Furthermore, we have found the minimum element of $L(z)$ that belongs to a subtree on the right side of the subtree rooted at p and we stored in w a pointer to that element (y). If w was a destination node before it became a node of the leftmost path, this information was stored during the actions for future use of case 3 for backward steps. Otherwise, the information was stored during the actions for future use of case 1 or case 2 for forward steps. Now we can make the necessary changes in O(1) time. Because the number of w's children is reduced by one, the pointer to p is replaced by a pointer to the node on the left side of p, which is the left neighbor of p in $LS(w)$. Let f be that element. *This change may cause a change to the pointer to y* (this may not happen at another level, if no node was deleted at the level below). If this pointer needs to be updated, it will point at the elements two positions left from y. Let g be that element. We split the lists $LS(z)$ and $L(z)$ at the elements f and g. The elements belong to the right parts. The right part of $LS(z)$ is connected to $LS(w)$. The new list is $LS(w)$. The right part of $L(z)$ is connected to $L(w)$. The new list is $L(w)$. We update the min-max pointers of z, w.
- $|w| + |z| < 2b$. Let p be the child of z such that the number of w's children on the left of p, is b. For p and every one of z's children on the right side of p we have set their $father_1$ and $father_2$ pointers to z, w respectively. We have also stored in w a pointer (*LS_cut_pointer*) to p. Furthermore, we have found the minimum element of $L(z)$ that belongs to a subtree on the right side of the subtree rooted at p and we have stored in w a pointer to that element (y). If w was a destination node before it became a node of the leftmost path, this information was stored during the actions for future use of case 3 for backward steps. Otherwise, the information was stored during the actions for future use of case 1 or case 2 for forward steps. We split the list $LS(z)$ and $L(z)$ at the elements p and y. The elements belong to the right parts. The right part of $LS(z)$ is connected to $LS(w)$. The new list is $LS(w)$. The right part of $L(z)$ is connected to $L(w)$. The new list is $L(w)$. We update the min-max pointers of z, w. Then we make z a pointer to w. We connect $LS(z)$ with $LS(w)$. The new list is $LS(w)$. We connect $L(z)$ with $L(w)$. The new list is $L(w)$. Node w now takes the place of z.
- $|w| > b$ (there is a pointer-node pointing to w). This case is introduced as a result of making a node, a pointer to another node. From the actions described so far, we make the following observation. *A node that is going to become a pointer to another node, has only b children.* In Fig. 13, z is a pointer-node, that points to w. This means that the lists $LS(z)$ and $L(z)$ do not exist, but some of the children of w are able to access w via z. Let r be the node on the left side of u_{j_1}. If r has a pointer to w, then we do nothing. If r has a pointer to z we delete w and restore z, as a valid node of the tree. $LS(w)$ becomes $LS(z)$, $L(w)$ becomes $L(z)$ and $father(w)$ becomes $father(z)$. z is now a node of the rightmost path of the tree, but since it has b children, it does not need a *candidate_right* node. All these changes cost O(1).

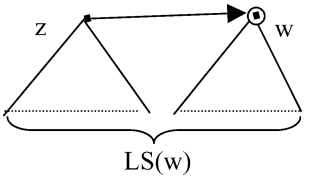

Fig. 13.

In Fig. 13, z is a pointer-node, that points to w. This means that the lists $LS(z)$ and $L(z)$ do not exist, but some of the children of w are able to access w via z. Let r be the node on the left side of u_{j_1}. If r has a pointer to w, then we do nothing. If r has a pointer to z we delete w and restore z, as a valid node of the tree. $LS(w)$ becomes $LS(z)$, $L(w)$ becomes $L(z)$ and $father(w)$ becomes $father(z)$. z is now a node of the rightmost path of the tree, but since it has b children, it does not need a *candidate_right* node. All these changes cost O(1).

Before we walk to the above level, we update the *min-max* pointers of all the affected nodes.

If u_2 falls into case 2, we join u_2 with w. This is done in O(1) time by making w a pointer to u_{j_1}. In this case, two nodes have to be deleted from that level. This fact does not change the logic described above. We can still process every one of the above levels in O(1) time. Also, in this case *the node that is going to become a pointer has more than b children. We allow this to happen, since the pointed node is the root of the tree.*

Following the logic described above we walk upwards and finally we process the level of v.

Finally if c is equal to 1 we extract a tree with root u_2, otherwise we do not extract such a tree.

End of procedure Tree_Extraction(u_1, u_2, c)

So in order to process the sequence of nodes $v, u_{j_1}, u_{j_2}, \ldots, u_{j_L}$ we just have to execute the procedures: *Tree_Extraction*$(v, u_{j_1}, 1)$, *Tree_Extraction*$(u_{j_1}, u_{j_2}, 1), \ldots,$ *Tree_Extraction*$(u_{j_i}, u_{j_{i+1}}, 1), \ldots,$ *Tree_Extraction*$(u_{j_{L-1}}, u_{j_L}, 1)$. With every tree created by this procedure, we associate the *dept list* that corresponds to the tree (a part of the dept list of T_v). Every time a tree $T_{u_{j_i}}$ is created from a tree $T_{u_{jk}}$ we scan the remaining list (which, initially is the entire dept list of T_v) and cut the part that corresponds to $T_{u_{j_k}}$. Starting from the beginning of the remaining list we find an element e that is larger than *max-pointer*(u_{j_k}). Let x be the left neighbor of e in the list. Then the dept list of $T_{u_{j_k}}$ is the list from the beginning until element x. The remaining list starts from e.

Since the processing of each tree level, during the *Tree_Extraction*() procedure can be performed in constant time we get that the time cost of procedure *Tree_Extraction*$(u_{j_i}, u_{j_{i+1}}, c)$ is d_i where d_i the distance between $u_{j_i}, u_{j_{i+1}}$. So, since the time cost of the tree separation step forms a telescopic sum we have the following lemma:

Lemma 11. *The time cost of the tree separation step is equal to* O(H).

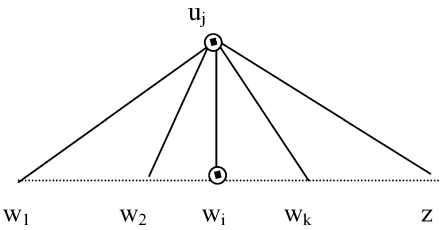

Fig. 14.

6.4.2. The root separation step

Now we concentrate on $T_{u_{j_L}}$. If $T_{u_{j_L}}$ has a non-empty *dept list*, we are going to create a new tree which does not have a *dept list*. In order to achieve this goal, we perform the root separation step on $T_{u_{j_L}}$.

Let $w_1, w_2, \ldots, w_k$ be the children (from left to right) of u_{j_L} that have positive counter (Fig. 14). The counter of node z is definitely zero because all the nodes from the rightmost leaf of the tree up to z, fall into case 5. From $T_{u_{j_L}}$ we create two new trees. The root of the first tree has as children all the children of u_{j_L}, from w_1 to w_k. The *dept list* of this tree is the *dept list* of $T_{u_{j_L}}$. The root of the second tree has children all the remaining children of u_{j_L}. Let u_f be that root. Then, Tu_f has no *dept list*. This happens because no separation step from any leaf of Tu_f has reached u_f. Tu_f meets Requirement 1 and is going to be the left part of the union operation. If $T_{u_{j_L}}$ is going to be the right part of a union operation, the root separation step is symmetrical. Thus:

Lemma 12. *The time cost of the root separation step is equal to* $O(b + H)$.

6.4.3. The complete algorithm for the union operation

Suppose we want to perform the operation $Union(q, s)$, where q and s are leaves of the trees T_u, T_w respectively.

We perform the tree separation step for T_u and create a new tree Tu_{j_L}. If $L(u_{j_L})$ is empty, then Tu_{j_L} is the left part of the union operation. If $L(u_{j_L})$ is non-empty, we perform the root separation step for Tu_{j_L} and create the tree Tu_f which is going to be the left part of the union operation. Let z be the rightmost leaf of the left part of the union operation, and f be its root (f is either u_{j_L}, or u_f). For T_f Requirement 1 is met. We perform the tree separation step for T_f without extracting z (we execute $Tree\_Extraction(f, z, 0)$). The nodes of the rightmost path of T_f are now heavy (Requirement 2 is met) and are not affected by the strategy.

We perform the tree separation step for T_w and create a new tree Tw_{j_k}. If $L(w_{j_k})$ is empty, then Tw_{j_k} is the right part of the union operation. If $L(w_{j_k})$ is non-empty, we perform the root separation step for Tw_{j_k} and create the tree which is going to be the right part of the union operation. Let r be the rightmost leaf of the right part of the union operation and e be its root (e is either w_{jk} or w_g). For T_e Requirement 1 is met. We perform the tree separation step for T_e without extracting r (we execute $Tree\_Extraction(e, r, 0)$). The nodes of the rightmost path of T_e are now heavy (Requirement 2 is met) and are not affected by the strategy.

In order to join the trees T_f and T_e, we use a slightly modified version of the union operation of Algorithm 1. The only modification for Algorithm 1, is that the number of children of a node, is the number of its children which are destination nodes. It is obvious that the changes to be made on the union operation of Algorithm 1, are minor and do not affect the logic and the time complexity of the operation.

Remark 1. If we apply the tree separation step for a tree, and the counter of the rightmost (leftmost) leaf of the tree (let x be that leaf) is at least two, then we produce a tree whose root is node x. If the counter of x is 1, let u_{j_L} be the node of the rightmost (leftmost) path of the tree with the maximum depth. The tree separation step produces a tree rooted at u_{j_L}. We apply again the tree separation step on Tu_{j_L} (*Tree_Extraction*$(u_{j_L}, x, 1)$) and create a new tree rooted at x. The total cost in order to produce a tree rooted at x is $O(b + H)$. Therefore, Target 2 for Algorithm 2 is now achieved.

Remark 2. Up to now we have seen that in some cases we transformed a node into a pointer. This transformation is a rather strange solution, forced by our need to process each level in O(1) time. The question is what kind of problems does this transformation create. Is it possible that it would lead to trees with height more than $O(H)$? We can distinguish two different cases:

- The node points to the root. In this case the node may have more than b children. The deletion of this node can be done for every tree, when a separation step is going to be performed for that tree. Therefore we add this case into the *undone job* for a tree. The cost for repairing this problem is $O(b)$. So the total cost for the undone job for a single tree is $O(b + H)$.
- The node does not point to the root. In this case the node did not belong to the leftmost, or to the rightmost path of a tree. We will show that the existence of these pointer-nodes does not create any problems. The maximum distance from the root to a leaf is now $2h$, where h would be the distance if no pointers existed. When the leaf separation steps visit a pointer-node or a node that is pointed by a pointer-node, the pointer-node is deleted because the time for performing this task can be spared at that point. The existence of pointer-nodes does not create any problems for the union algorithm of Algorithm 1. At the first execution of procedure *Union_Level* (phase 1) we can spare $O(b)$ time, so if we find a pointer-node that points to u (see Algorithm 1), we have the needed time to delete the pointer-node. The same holds for the last execution (phase 3), since we can also spare $O(b)$ time. If in phase 2, the node of T_A (let w be that node) is pointed by a pointer-node (let z be that node), then we are still able to process each level in O(1). If we are going to add a new child to such a node, we add the new child and:
- If w has $2b$ children, we restore z, as a valid node of the tree. This can be done in O(1) time. After restoring z we delete w from $LS(r)$ (r is the father of w) and proceed to the above level by executing *Union_Level*$(r, \{w\})$ (r is the father of z).
- Otherwise, we do nothing and proceed to the above level by executing *Union_Level*$(r,$ *nil*$)$ (r is the father of w).

By combining the findings of Lemmas 11, 12 and the aforementioned remarks we get the following theorem:

Theorem 1. *The time cost of the Union procedure is equal to* $O(b + H)$*, where* b *is the branching factor and* H *is the maximum height of an L_Tree.*

7. The forest of *L_Equivalent* trees

The careful reader should have noticed that a problem that can arise from our solution to the empty leaves problem, as given in Algorithm 2, is the creation of trees containing no separation leaves. This implies that it could be possible to have the creation of long chains of such trees thus spoiling the time complexity of our search procedure. In the sequel we will prove that the number of consecutive *L_Trees* having no separation leaves can be at most 2 and so the time complexity of the procedure remains reasonably small.

Definition 5. We call an *L_Equivalent* tree empty if it has no separation leaves. If u is the root of an empty *L_Equivalent* tree, then the list $L(u)$ is empty.

Definition 6. We call two *L_Equivalent* trees adjacent, if the predecessor (successor) end-node of the leftmost (rightmost) leaf of the one tree, is the rightmost (leftmost) leaf of the other.

It is not difficult to see that empty trees may be created as a result of the operations introduced so far. We have to find a way to bound the number of the empty adjacent trees, to a constant factor. This way, given an empty leaf, the name of the set that contains x (the closest filled leaf in the left direction) is only a constant number of trees far. What we need is the following restriction to hold:

Emptiness Restriction. *Let* T_u, T_w *be two adjacent L_Equivalent trees. Then, at most one of the two trees is empty.*

A possible empty tree can be created by the following actions:

1. In the tree separation step, suppose that u_{j_1} falls into cases 2, 3 or 4. This means that the tree rooted at u_{j_1}, created by the algorithm, may be empty while the tree rooted at v definitely is non-empty since node w (next to u_{j_1}) is affected by some leaf separation step (which means that in the subtree rooted at w there is at least one separation leaf). If u_{j_1} falls into case 1, then the tree rooted at u_{j_1} definitely is non-empty since it has at least one separation leaf. Let us consider the first of these two cases, that is when T_v may be empty. Suppose that the adjacent tree of T_v on the left side (T_w) is empty. Initially (before the tree separation step) the emptiness restriction holds. When the tree rooted at u_{j_1} is extracted, T_v may become empty. As a consequence, the adjacent trees T_w, T_v may both be empty, which means that the emptiness restriction is violated. To prevent this from happening, we must check during the tree separation step if the

leftmost (rightmost in the symmetrical case) tree is empty (the list $L(v)$ is empty). If so, we check its adjacent tree (T_w) which was not created by tree separation step. If T_w is also empty, we use the union operation of Algorithm 1 to join the two trees.

2. In the root separation step, the created tree to be used by the union operation of Algorithm 1 may be empty while the other created tree is definitely non-empty.
3. In the top-down forward separation step (see Fig. 10), T_{u_1} (or T_{u_2}) may be empty. If T_{u1} is empty and its adjacent tree on the left side (T_y) is empty, the emptiness restriction is violated, and we have to perform $Union(T_y, T_{u_1})$. If T_{u_2} is empty and its adjacent tree on the right side (T_z) is empty, the emptiness restriction is violated, and we have to perform $Union(T_{u_2}, T_z)$. The above actions are performed during the top-down forward separation step. The reason why they were not mentioned at that point, is that the union operation was not yet introduced.
4. Suppose the counter of a leaf x is reduced and becomes zero. If x was the only leaf of its tree with positive counter, then the tree becomes empty. Let u be the root of the tree (the root may also be x) and let w be the root of the adjacent tree of T_u on the left side and z be the root of the adjacent tree of T_u on the right side. If T_w is empty, we perform union (T_w, T_u). Let r be the root of the new tree. If T_z is empty, we perform $Union(T_r, T_z)$. If T_w was not empty we perform $Union(T_u, T_z)$.
5. A leaf structure of an end-node x, which is the root of an *L_Equivalent* tree, is split into two parts, one of which is empty. Let w be the new empty leaf and z be the new filled leaf. Node w is now a new empty tree. Let r be the root of its adjacent tree on the different direction than z. If T_r is empty, we perform $union(T_w, T_r)$.

Lemma 13. *We have a forest of L_Equivalent trees and a union operation is performed. If the emptiness restriction holds before the union operation then it holds after the union operation also.*

Proof. The lemma follows from actions 1, 2. □

Lemma 14. *We have a forest of L_Equivalent trees and a top-down forward separation step is performed. If the emptiness restriction holds before the separation step then it holds after the separation step also.*

Proof. The lemma follows from action 3. □

Lemma 15. *We have a forest of L_Equivalent trees and a bottom-up backward separation step is performed. If the emptiness restriction holds before the separation step then it holds after the separation step also.*

Proof. The lemma follows from action 4. □

Lemma 16. *We have a forest of L_Equivalent trees and a new empty tree is created when the leaf structure of a filled leaf is split. If the emptiness restriction holds before the creation of the new empty tree then it holds after the creation of the new empty tree also.*

Proof. The lemma follows from action 5. □

Theorem 2. *Let I be the forest of the L_Equivalent trees and a dynamic operation $(Insert(x), Delete(x))$ is performed. If the emptiness restriction holds for I before the operation then it holds after the operation also.*

Proof. It follows from Lemmas 13–16. □

Theorem 3. *Let I be the forest of L_Equivalent trees. The emptiness restriction for I holds at any time.*

Proof. Let I_1 be the present instance of the forest. Suppose that the emptiness restriction does not hold for I_1, which means that there are two empty adjacent trees, T_v, T_w. Let x_1 be the leaf that became empty first of all the other leaves of T_v, T_w. Let I_2 be the instance of the forest just before x_1 becomes empty. If the emptiness restriction holds for I_2, it holds for I_1 as well (Theorem 2). Therefore, the restriction does not hold for I_2 and let T_{v_2}, T_{w_2} be two empty adjacent trees of I_2. Let x_2 be the leaf that became empty first of all the other leaves of T_{v_2}, T_{w_2}. If we continue to apply the same logic in order to move back in time, we will finally reach the instance I_{initial} of the forest where x_1 is going to become the only empty leaf in the base structure. It is obvious that the restriction holds for I_{initial}, and as a consequence (Theorem 2), it must hold for I_1 as well, because I_1 is the result of performing a number of dynamic operations on the instance I_{initial}. □

8. The complete algorithm for the dynamic predecessor problem

In the sequel we will present the complete algorithm for the dynamic predecessor problem incorporating in its description both Algorithm 2 and the efficient manipulation of the leaf structures. As it can be seen from the pseudo-code that follows, each basic operation is implemented in a constant number of operations each of which costs $O(\log\log|U|/\log\log\log|U|)$ time, with the exception of the search operation that has time cost equal to $O(\log\log|U|)$. Let us now consider the space complexity. Since the number of nodes in the trie structure is $O(|U|/\log|U|)$ and each element is stored once, the space complexity is $O(|U|/\log|U|+n)$. This space complexity can be reduced to linear if we employ *dynamic perfect hashing* [8]. In particular, for each level of the trie we store the red nodes in a dynamic perfect hash table. Since the number of red nodes is at most the number of stored elements we get that the total space complexity is linear; the time complexity of the query and update operations remain invariant with the exception that the bounds in the update operations are expected amortized.

So, we get our final theorem:

Theorem 4. *There exists a data structure for the dynamic predecessor problem that uses $O(|U|/\log|U|+n)$ space, performs all the operations in $O(\log\log|U|)$ time and needs $O(\log\log|U|/\log\log\log|U|)$ structural changes per update operation. In particular the time cost of each update operation is $O(\log\log|U|/\log\log\log|U|)$ provided that we have*

access to the update position. The space complexity becomes linear if we employ randomization. In this case the time complexity of the query and update operations remain invariant with the exception that the bounds in the update operations are expected amortized.

Pred(x) Find the last red node (the end-node) of the root to x path. Let u be that node.
If u is a filled leaf, let A be the leaf structure of u. Perform $Pred_L(A, x)$.
If x is smaller than the smallest element in A, then use the doubly linked list and go to the predecessor end-node of u. Let w be that node.
If w is a filled leaf, then the maximum element of the leaf structure of w is the predecessor element of x.
Else if w is an empty leaf find the closest filled leaf in the left direction (due to the emptiness restriction we have to visit a constant number of *L_Equivalent* trees). Return the maximum element of the leaf structure of that leaf.
Else if u is an empty leaf find the closest filled leaf in the left direction (due to the emptiness restriction we have to visit a constant number of *L_Equivalent* trees). Return the maximum element of the leaf structure of that leaf.

Insert(x) Find the end-node of the path from the root to x. Let w be that node and i be the depth of w.
$Insert_L(A, x)$ where A is the leaf structure of w.
If after the insertion the number of stored elements becomes one, then $Split(q, w)$, where q the *L_tree* where w belongs.
Else if after the insertion the number of stored elements becomes $\log\log|U| + i + 1$, then
Open(w).
$Split_L(A, y)$ where $y = \langle p\rangle 10\ldots0$. After the split two new leaf structures, A_1 and A_2, are created.
Store the pointer to A_1 into the left child of w.
Store the pointer to A_2 into the right child of w.
Replace w by its children in the doubly linked lists of the end-nodes.
If A_1 is empty then:
Create an *L_Tree* with unique element the left child of w.
Unite this tree with the *L_Tree* left of w.
If A_2 is empty then
Create an *L_Tree* containing the two children of w.
Unite this tree with the *L_Tree* right of w.
Else if w is not the root of an *L_Equivalent* tree then
Perform the undone job for the tree that contains w.
If w is in bottom-up phase then
If the counter of w is greater than the height of the root of the tree or equal to the height of the root of the tree then
w enters the top-down phase.
Perform a top-down forward separation step for w.
Else Perform a bottom-up forward separation step for w.

Fi.
Else Perform a top-down forward separation step for w.
Fi
Fi

Delete(x) Find the end-node of the path from the root to x. Let w be that node and i be the depth of w. Let u be the brother of w and r be the father of u, w. If u is an end-node, let Z be its leaf structure.
$Delete_L(A, x)$, where A is the leaf structure of w.
If u is an end-node and the sum of the elements of the leaf structures A and Z becomes $\log\log|U| + i - 1$ after the deletion of x, then
Close(u, w).
$Union_L(A, Z)$. A new leaf structure B is created.
Store the pointer to B into the father of w.
If w belonged to an *L_Equivalent* tree (T_1) then
Create an *L_Equivalent* tree with root w.
Fi
If u belonged to an *L_Equivalent* tree (T_2) then
Create an *L_Equivalent* tree with root u.
Fi.
Delete u, w from the doubly linked list of the end-nodes, and replace them by r.
Else if w is in bottom-up phase then
If after the deletion leaf structure A does not contain any elements then
Perform the inverse procedure of *Split*(q, w), where q the *L_tree* where w belongs.
Else perform a backward bottom-up separation step for w.
Fi.
Else (w is in top-down phase) then
If leaf structure A contains i elements, and the height of the *L_tree* that contains w is $j \leqslant i$ then do nothing
Else
w returns to bottom-up phase
Perform a backward bottom-up separation step for w.
Fi.
Fi.

9. Conclusions

For the dynamic predecessor problem, searching along the path of the trie structure (van Emde Boas tree) is the main obstacle that needs to be overcome. In the static case, Beame and Fich [5] presented a static solution to the problem that reaches the lower bound. This bound is not yet reached for the dynamic case.

In this paper we divided the predecessor problem into two sub-problems:

- Searching along the path of the base structure;
- Whatever comes afterwards;

and we managed to solve the second sub-problem in $O(\log\log|U|/\log\log\log|U|)$ worst-case time. Solutions that accomplish better time results can possibly be found, by using the van Emde Boas' tree and bucketing at the lower levels. The main novelty in the presented structure is that the root to leaf path is now suitable for using a better strategy than binary search. This way we leave the field open for finding a better solution than binary search for searching along the path.

References

[1] A. Andersson, Sublogarithmic searching without multiplications, in: Proc. of the 36th Annual Symposium on Foundations of Computer Science (FOCS), 1995, pp. 655–663.
[2] A. Andersson, Faster deterministic sorting and searching in linear space, in: Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS), 1996, pp. 135–141.
[3] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time?, in: Proc. 27th Annual Symposium on Theory of Computing (STOC), 1995, pp. 427–436.
[4] A. Andersson, M. Thorup, Tight(er) worst-case bounds on dynamic searching and priority queues, in: Proc. of the 32th Annual Symposium on Theory of Computing (STOC), 2000, pp. 335–342.
[5] P. Beame, F.E. Fich, Optimal bounds for the predecessor problem and related problems, J. Comput. System Sci. 65 (1) (2002) 38–72.
[6] N. Blum, On the single-operation worst-case time complexity of the disjoint set union problem, SIAM J. Comput. 15 (1986) 1021–1024.
[7] G.S. Brodal, Finger search trees with constant insertion time, in: Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, pp. 540–549.
[8] M. Dietzfeldbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Ronhert, R. Tarjan, Dynamic perfect hashing: upper and lower bounds, SIAM J. Comput. 23 (1994) 738–761.
[9] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. System Sci. 47 (1993) 424–436.
[10] M.L. Fredman, D.E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, J. Comput. System Sci. 48 (1994) 533–551.
[11] K. Mehlhorn, Data Structures and Efficient Algorithms, vol. 1: Sorting and Searching, Springer-Verlag, Berlin, 1984.
[12] K. Mehlhorn, S. Naher, H. Alt, A lower bound on the complexity of the union-split-find problem, SIAM J. Comput. 17 (6) (1988) 1093–1102.
[13] M.H. Overmars, Computational geometry on a grid: an overview, in: NATO ASI Series, vol. F40, 1988, pp. 167–184.
[14] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, Inform. Process. Lett. 6 (1977) 80–82.
[15] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Systems Theory 10 (1977) 99–127.
[16] D.E. Willard, Examining computational geometry, van Emde Boas trees and hashing from the perspective of the fusion tree, SIAM J. Comput. 29 (3) (2000) 1030–1049.